

GRAVITY GROOVE

Artificial Intelligence Document
developed by



**SPACE
MUFFIN**

Index

Space Muffin Team	2
1 General Considerations	3
2 Movement Algorithms	3
2.1 Description of Desired Movement	3
2.2 General Purpose Approach with Physics	3
2.2.1 Gravity Implementation	3
2.2.2 Movement Implementation	5
2.2.3 Edge Detection	5
2.2.4 Rotation	6
2.3 Custom Approach Knowing Platforms Shapes	7
2.4 The Use of Physics on the Network	9
2.4.1 Rewind and Replay Algorithm	9
2.5 The Reasons of our Choice	10
3 Planning & Pathfinding	10
4 NPCs' Behaviour	11
4.1 Chaser Bot	11
4.1.2 Policeman Bot	13
4.1.3 Gold Magnet Bot	14
4.2 Missile Turret	15

Space Muffin Team



Giulia Barnaba
Game Designer
giulia.barnaba92@gmail.com



Lorenzo De Simone
Game Designer
Game Programmer
lorenz.desimone@gmail.com



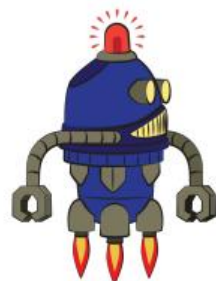
Andrea Jegher
Game Programmer
andrea.jegher@mail.polimi.it



Emanuele Lattanzio
Game Designer
Game Programmer
emanuele.lattanzio@studenti.unimi.it



Marta Barducchi
Visual Artist
marta.barducchi@gmail.com



Valerio De Vittorio
Sound Designer
Music Composer
valky3it@gmail.com



1 General Considerations

Gravity Groove features a peculiar movement mechanic, hence a custom usage of known AI techniques for planning was necessary in order to produce the desired results.

While A* and basic graph theory was used for the aforementioned planning, the main challenge was to produce a polished movement algorithm mixing real physics simulation with kinematic movement.

2 Movement Algorithms

The algorithms we are going to describe are used for the movement of nearly every character of our game, both players and NPCs.

2.1 Description of Desired Movement

Players and NPCs need to move around planets and platforms regardless of their shape and one of the main focuses of our efforts was to keep the character always and completely glued to the surface, while being able to jump and having a physics-like feel only when in air.

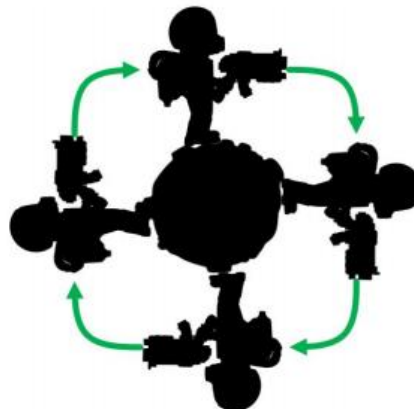


Fig.1 - Movement around a planet

After an extensive research and many prototypes, we formalized two approaches to the problem worth considering, that are described in chapters 2.2 and 2.3. In the chapter 2.5 we will explain why we preferred to use the first method over the second after describing them in detail.

2.2 General Purpose Approach with Physics

2.2.1 Gravity Implementation

This method is engineered to work without knowing the shape of the underlying planet.

First of all, standard gravity is completely removed and a custom gravity vector is used.

The character has a trigger around him that registers each platform he can walk on that is nearby, as shown in picture. The closest inside this trigger will be referred as the **ground** of the player from now on.

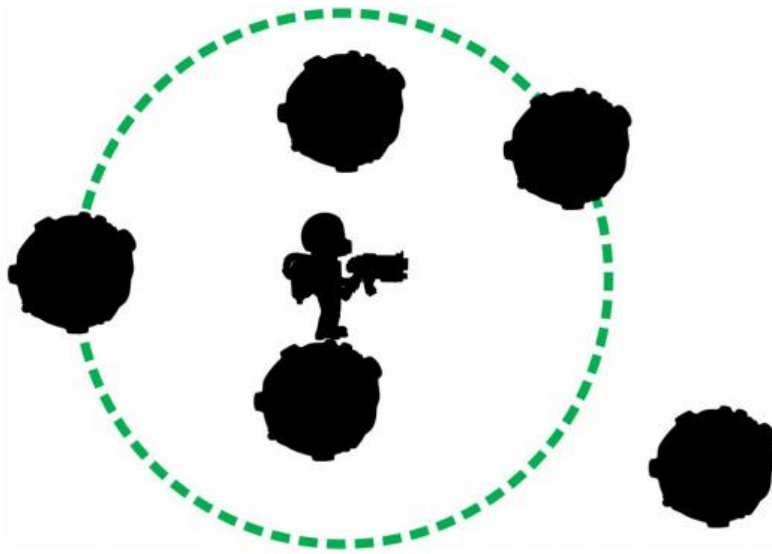


Fig.2 - Player radar for nearby planets

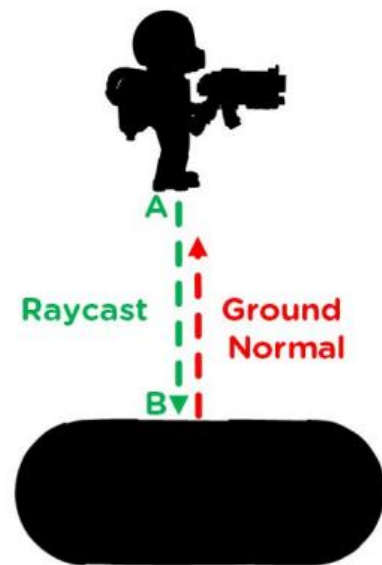


Fig.3 - Raycast and gravity versor

Given a position of the character, it is always possible to draw a raycast beneath his feet in order to reach the current ground as shown in Figure 3.

After a study of the state of the art techniques used in other games such as *Super Mario Galaxy*, an approach based on ground normal and raycast emerged as a possible solution.

Given point A and B, respectively on the character's collider and the ground's collider, the AB vector gives the shortest possible vector between the two colliders. This vector is used in order to raycast towards the ground and obtain the ground normal in that point. Once that normal is obtained, a force is applied in the opposite direction, making the character attracted towards the ground.

It is important to state that in physics, the attraction would naturally be towards the platform center rather than the antinormal of the ground; on platforms that are not perfect circles though, this would produce an undesired effect during movement as shown in Figure 4. During movement, the character would be “held back”, reducing his freedom.



Fig.4 - Wrong gravity towards center

Jump implementation is fairly easy; when a character has to jump, an impulse force is given following the normal of the ground. It is quite obvious that the module of the jump vector should be higher than the gravity one.

2.2.2 Movement Implementation

Given the attraction forces, we can now define how the character movement on the platforms is defined. In order to produce a clean movement, the movement vector should always be perpendicular to the ground normal.

Since we are in a 2D space, for each vector (x,y) there are two possible perpendicular vectors, $(-y,x)$ for counterclockwise movement and $(y,-x)$ for clockwise movement.

Given player input, it is possible to use one of the aforementioned vectors, multiply them by the speed of the character and make him move.

Applying forces using physics in this case would be quite problematic: in this particular game we want player to have perfect control over his character. Applying forces instead we would see characters suffering from inertia and detaching from the ground.

This is the reason why the algorithm used for movement on the ground is purely kinematic. The next position of the character is simply translated of a vector that is perpendicular to the normal ground and multiplied by a speed factor.

This led to a quite satisfactory result with a corner case that needed polishing: while moving on edges, the movement wasn't clean. The character always moved beyond the limits of the platform, resulting in an unpleasant feeling for the player.

That is the reason why a custom edge detection algorithm has been developed to correct the velocity vector when the character is moving on an edge of a platform.

2.2.3 Edge Detection

In order to detect an edge, we use this custom method.

First of all we calculate the next position using the method described in the previous chapter. Before moving the character there, a ray is cast down in order to see if the ground is there. If the raycast finds a ground, then the character is moved to the new position since there is no edge.

If the raycast doesn't find a ground, this point is used for another raycast in the opposite movement direction, in order to find a point on the surface of the ground. Once that point is found, it is elevated to match the character's height. The difference between this newly found point and the character position, normalized and multiplied by the character speed, is the corrected movement vector.

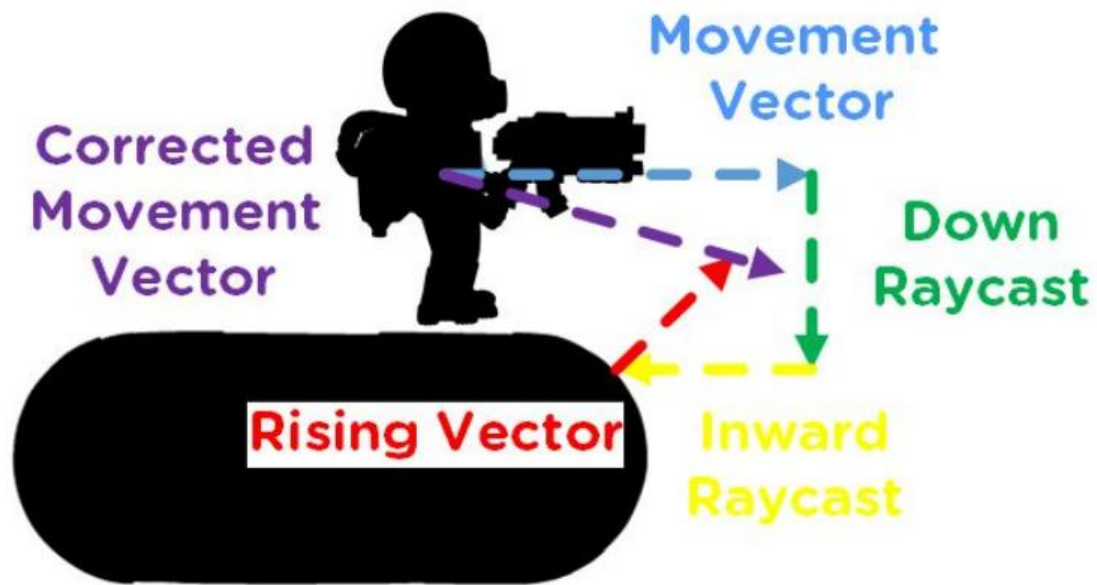


Fig.5 - Edge detection algorithm

This way, the character always sticks to the surface and the movement is perfectly kinematic. It is important to note that this algorithm doesn't know anything about the ground shape and therefore is ready to be used for new platforms with different shapes.

2.2.4 Rotation

Rotation is done using a Lerp function that takes account of the normal of the ground, with the local up vector lerp towards it.

In order to cover all the possible cases in a clean way and avoid gimbal locks, we decided to interpolate quaternions.

This approach led us to a smooth rotation both when changing planets in the air and when moving on platforms.

2.3 Custom Approach Knowing Platforms Shapes

In order to simplify the problem we tried to use a geometrical approach, so we divided the platforms and planets with geometric shapes easy to use.

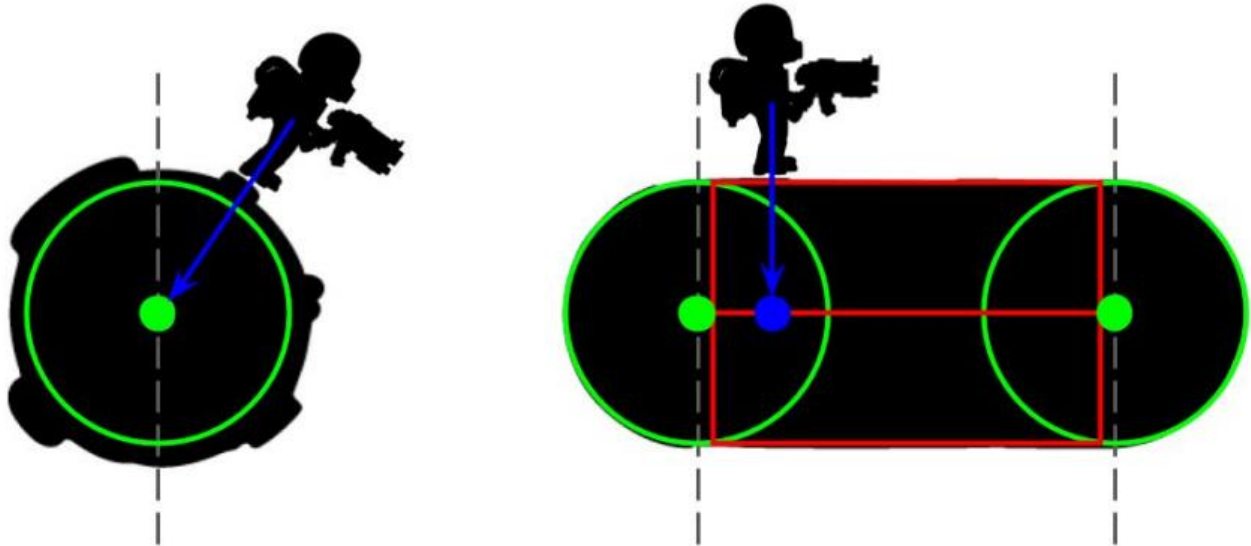


Fig.6 - Attraction points

The main problem was to find the point of gravity attraction to avoid the problem of attraction on the center of the platform explained on point 2.2.1 Fig. 4.

With only two points we could divide the platform in three geometrical parts, two circles with the center of it that is the attraction point and a rectangle with center of attraction that have x equal to the player position and y equal to the platform center.

Once that problem is resolved we worked to remove physical forces for jump and movement.

We divided the movement in three parts:

1. Move player towards his left/right
2. Rotate the player towards the attraction point
3. Apply gravity

All these tasks are needed to know the next position that the player will have, all of this scaled with velocity and delta time to have a fluid movement.

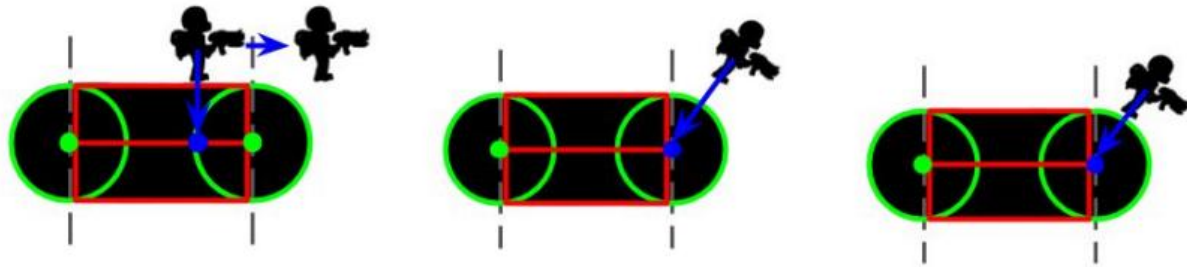


Fig.7 - Movement rotation

The resulting movement is faster on the middle section because the distance on the lateral circles is projected onto the point of the attraction, resulting in a shorter actual distance on the surface.

We resolved that problem using two different velocities.

The best implementation would be calculating geometrically the next position of the character on the circle so that the arc of circumference is equal to the perimeter he would cover if it were linear.

We implemented the jump moving the player on his up and multiply that vector with a velocity that decrease of a fixed value to simulate the gravity effect.

The best implementation is to decrease the velocity not to a fixed value but a more realistic formula like a gaussian.

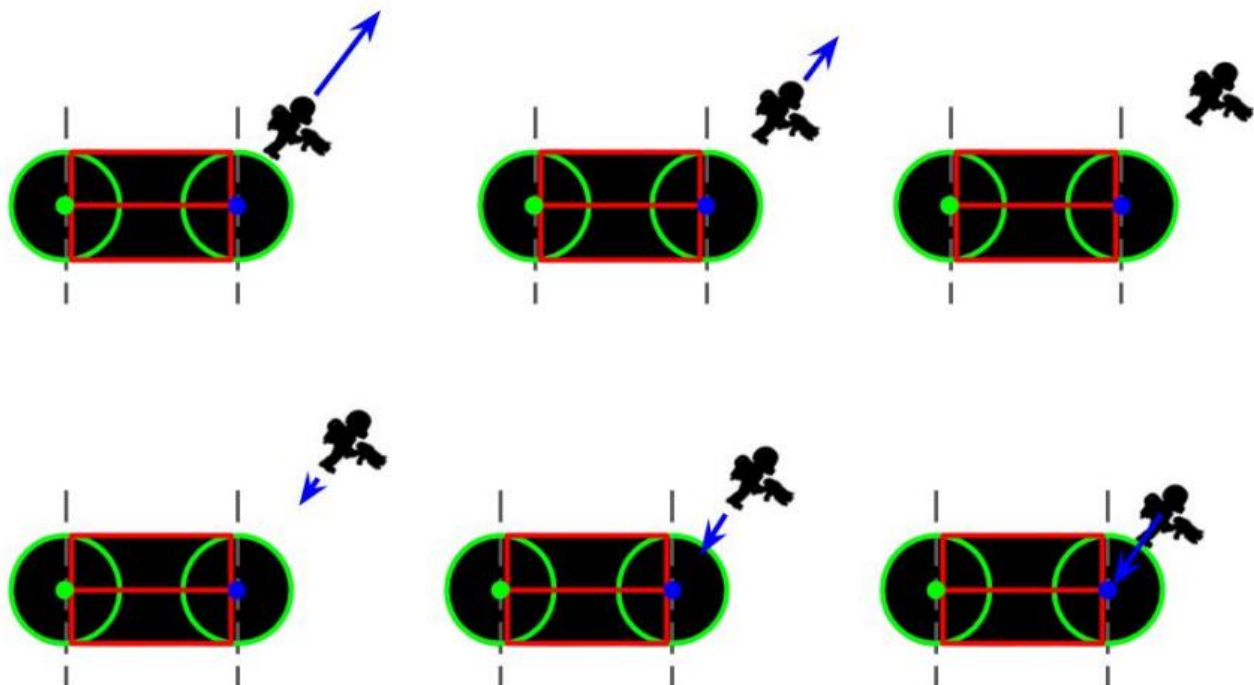


Fig.8 - Jump mechanic

The only thing left to do is the removal of rigidbody for collisions in order to completely remove Unity physics.

We used a method that checks the next distance between the feet of character and the center of gravity of platform; if that distance is lower than the radius of the circle or if the next position is on the other side of the platform, the next position of the feet's player is setted to collide with the radius.

2.4 The Use of Physics on the Network

One of the main problems of synching movement over the network is finding an algorithm that is perfectly reproducible both on server and client, in order to implement prediction and extrapolation.

Unity physics, unfortunately, isn't deterministic and isn't reliable in that regard.

The best approach is to reproduce a custom physics with Unity transforms and write a custom prediction and interpolation algorithm to predict client-side the movement while maintaining server authority. Given our time constraint to produce a playable prototype though, this approach would have required too much time and therefore we looked for quicker alternatives. As a matter of fact, Unity offers a built-in interpolation and prediction when syncing rigidbodies over the network; these features aren't offered when syncing directly the object transform only, resulting in an unpleasant lag even when the network delay is very small. Therefore we decided to sync rigidbodies and apply forces for attraction and jump, we tuned the parameters offered by Unity and we managed to find a good compromise between interpolation and prediction.

2.4.1 Rewind and Replay Algorithm

In order to further reduce data usage and reduce lag, we studied state of the art techniques used in AAA online multiplayer titles but, as mentioned before, the time constraints were too tight to implement them correctly.

The algorithm we wanted to implement if we had time in order to discard physics completely and reduce lag is called Rewind and Replay.

This algorithm requires a reliable network timestamp.

The main focus of this algorithm is keeping the server authority over the character movement and, at the same time, giving the player an immediate and local feedback on his input; basically, the best of both worlds.

This algorithm requires an **identical** and **deterministic** movement simulation running both on server and client (not doable using Unity physics).

When a character receives input from the local player, the input is rendered right away and is stored locally in a data structure with a network timestamp. This gives the player the immediate feedback on its input, as if he were playing offline.

A parallel thread fills this buffer with inputs and their timestamp and sends them to the server with a defined frequency.

The server reads them and executes them immediately in its context and, at the same time, sends the correct character position at a given timestamp.

When the local player receives the correct position from the server, the timestamp is obviously older and the position received is basically giving information on where he had to be in a given timestamp in the past.

Snapping the character directly to the position received by the server is a terrible choice since the player would see his character snapped to a past position each time a packet is received from server which is unacceptable.

Instead, when a packet is received from server, the client “virtually” resets to that position. This is the **rewind** part of the algorithm.

Then the client reads its local input buffer and for each input stored does the following things:

1. If the timestamp of the input is older than the position received by the server, discard it and remove it from the buffer.
2. If the timestamp of the input is younger than the position received by the server, execute it once again and update the virtual position. This is the **replay** part of the algorithm.

The virtual position is updated with each input stored in the buffer and then applied to the actual character position.

This way, the client has immediate feedback on his input and receives a delayed and disguised correction by the server. The best thing about this is that basically zero correction has to be done when client and server have the same simulation. In case of a hacked client, the server authority is present since no position is sent from the client to the server; the server simply executes the input received from the client, therefore all other clients receive reliable information about the character.

2.5 The Reasons of our Choice

In conclusion, after examining many approaches and a study of state of the art techniques used both for movement and lag mitigation, we opted for a simple and quick approach in order to produce acceptable results in short time.

We decided to sync rigidbodies, using the built-in network prediction and interpolation offered by Unity, and the algorithms shown in chapter 2.2 in order to maximize the level design flexibility in case of new shapes.

3 Planning & Pathfinding

The main difficulty of our planning is to understand when a nearby platform is reachable or not with a single jump.

We resolved that problem using a graph that connects all platforms with its adjacencies.

The nodes of the graph are the existing planets and platforms, while the edges needs to

be calculated.

We could use two approaches to identify adjacencies: calculate them on runtime or connect platforms during level design.

We chose to use the first approach because it allow more flexibility to change positions of platforms and variability of the map during the game.

In our approach, we calculate adjacencies between platforms at the start of the game because in our maps the positions of platforms are fixed and there aren't events that change the graph (for now). Another reason why we make that calculation at start is a computation complexity problem since making a check from each platform is an expensive operation, so it is better if we do it before the game starts.

Our graph is calculated checking the distance between all platforms and if the distance between two platforms is less than the sum of the distance covered with a jump and the planet attraction range, an edge between those two nodes is added to the graph.

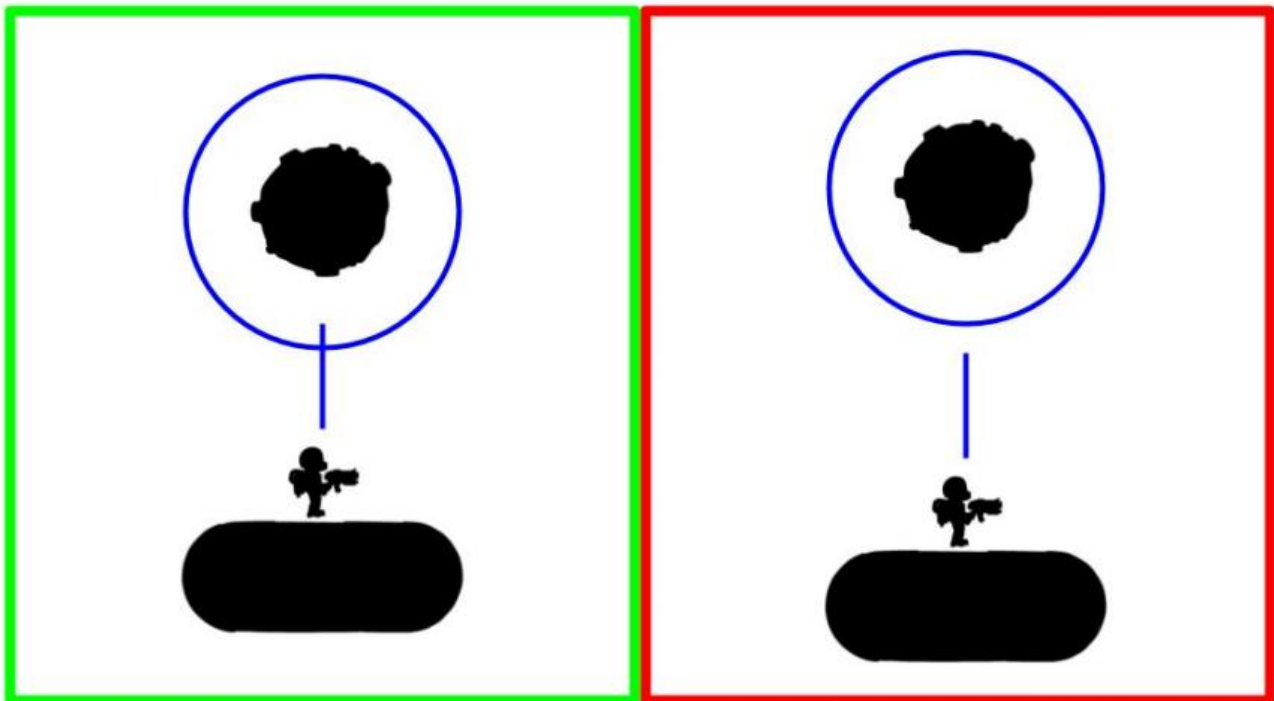


Fig.9 - Platforms adjacency

Given the structure of the graph, we can now examine when and how A* is used for planning and how it is linked to the movement algorithm.

All NPCs that need to move around planets use the movement algorithm described in the chapter 2. When an NPC chooses the planet it wants to reach, a coroutine starts to find the shortest path to reach it.

4 NPCs' Behaviour

4.1 Chaser Bot

This class of bots has a player as a target and tries to reach him all the time using A* for

planning.

While the pathfinding coroutine is running and the path isn't ready, the default action for this class of NPCs is simply to rotate around the current planet; this let us use a very sloppy timer (we used 0.2 seconds) for the pathfinding coroutine step that results in a very small amount of CPU used, which is particularly convenient since we are on mobile.

The chosen heuristic is the Euclidean distance, since there are no particular problems such as walls or obstacles in our game and provides a good estimation.

Once the path is ready, the agent knows what planet to reach next and casts a raycast over its head: if it finds the desired planet, it jumps and reaches it. In Figure 10, we can see the graph's edges in blue, and the aforementioned raycast in green.

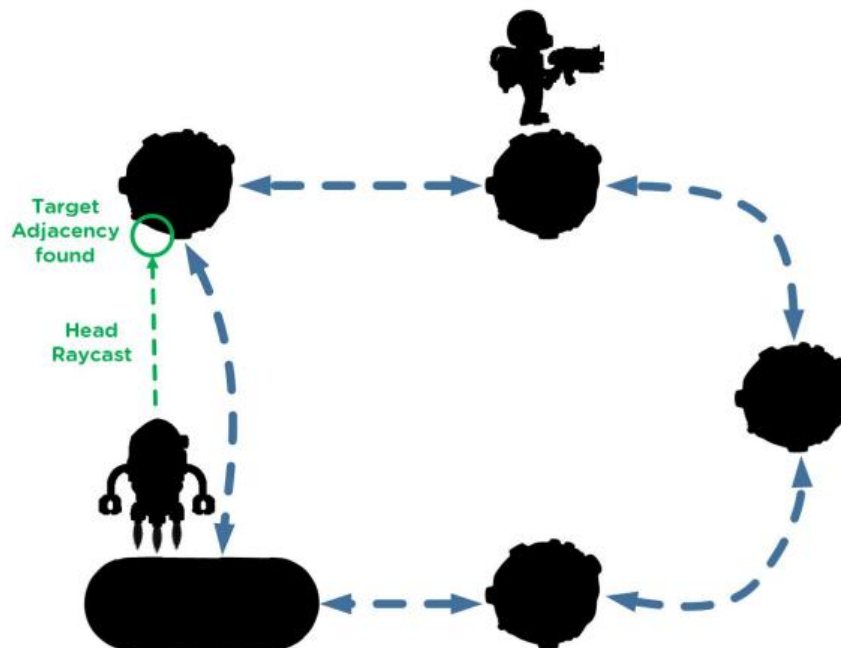


Fig. 10 How the agent discovers the right adjacency

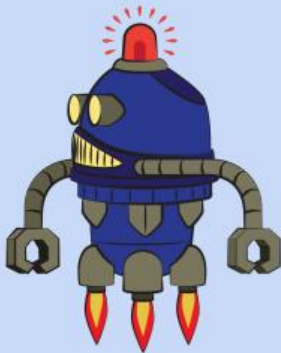
We decided to use the same script with this behaviour using delegate functions that each single bot will initialize accordingly. The delegates we identified are the following:

1. **Target Getter:** function that given the environment, returns a player.
2. **On Hit:** function that does something when the bot is hit by a missile.
3. **On Collectable Hit:** function that does something when the bot collides with a collectable item.

Simply modifying these functions gave us lots of freedom and made us produce two bots using the same pathfinding methods. It is important to state that this solution is flexible and is open to different delegate implementations in the future, hence new chaser bots are easy to implement.

4.1.2 Policeman Bot

Policeman Bot

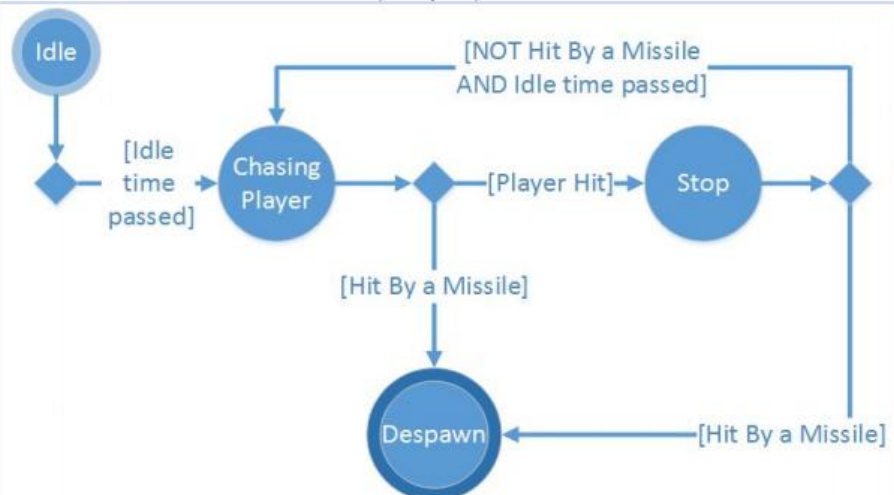


Behaviour

The Policeman Bot is always looking for some space rogues to hit. It patrols jumping from one planet to another hoping to find scoundrels to punish.

It moves and jumps like a player between planets. Its main focus is reach the player with the highest amount of coins in order to hit it. In order to do so, it chooses carefully the next planet in its path. When a player is hit, he/she drops coins as if it were it by a missile and the bot stands still for some time. The bot dies when is hit by a player missile.

Finite State Machine



- **Idle:** state when the bot spawns, does nothing and is harmless.
- **Chasing player:** state when the bot tries to reach the player with the highest point score.
- **Stop:** state when the bot just hit a player and stops for some seconds.
- **Despawn:** state when the bot just got hit by a missile and it despawns.

Target Getter

Gets the player with the highest amount of coins.

On Hit

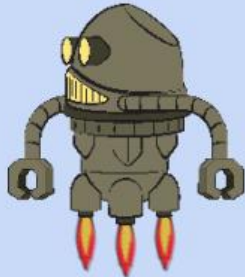
Simply despawns.

On Collectable Hit

Does nothing.

4.1.3 Gold Magnet Bot

Gold Magnet Bot



Behaviour

The Gold Magnet Bot is a bot that flies around planets and attracts coins from the ground.

Once spawned, it looks for the player with the lowest score. It decides reactively the best path to reach him/her and moves following it.

All the collectable coins in its area are drawn and stolen by it. When killed, it spawns all the coins collected so far that can be gathered by the players who pass by.

Finite State Machine



- **Idle:** state when the bot spawns, does nothing and is harmless.
- **Collecting coins:** state when the bot tries to collect coins while going near the player with the lowest score.
- **Stop:** state when the bot just hit a player and stops for some seconds.
- **Despawn:** state when the bot just got hit by a missile and it despawns. Before despawning, all coins collected by the agent so far are dropped.

Target Getter

Gets the player with the lowest amount of coins.

On Hit

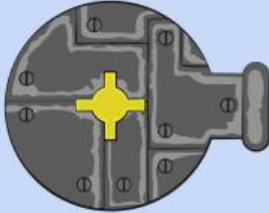
Deactivate sprite and collider until all coins are dropped, when despawn completely.

On Collectable Hit

Collects coin.

4.2 Missile Turret

Missile Turret

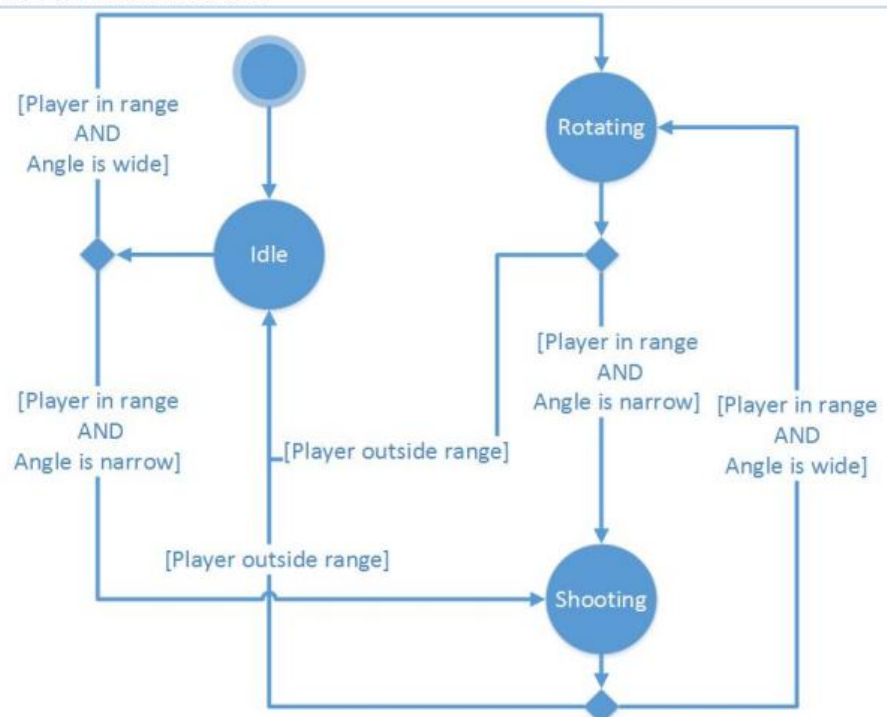


Behaviour

A missile turret that is crafted and designed to shoot to space rogues who dare to steal riches in the space.

The missile turret is floating in a fixed point in the air and when a player is in its range, it rotates towards him/her and it shoots a missile to hit him/her.

Finite State Machine



- **Idle:** state when turret waits still for some players to walk in its shooting range.
- **Rotating:** state when the turret finds a player within its shooting range and rotate towards him/her.
- **Shooting:** state when the turret shoots missiles towards a player that is within its shooting range and the angle is narrow.